

Lecture 11

Lists (& Sequences)

Using Try-Except

try:

```
result = input('Number: ')    # get number
x = float(result)              # convert to float
print('The next number is '+str(x+1))
```

except:

```
print('That is not a number!')
```

Similar to if-else

- But always does the try block
- Might not do **all** of the try block

Using Try-Except

try:

```
result = input('Number: ') # get number
x = float(result)           # convert to float
print('The next number is '+str(x+1))
```

Conversion
may crash

except:

```
print('That is not a number!')
```

Execute if crashes

Similar to if-else

- But always does the try block
- Might not do **all** of the try block

Try-Except is Very Versatile

```
def isfloat(s):
```

```
    """Returns: True if string  
    s represents a float"""
```

```
    try:
```

```
        x = float(s)
```

```
        return True
```

```
    except:
```

```
        return False
```

Conversion to a
float might fail

If attempt succeeds,
string s is a float

Otherwise, it is not

Try-Except and the Call Stack

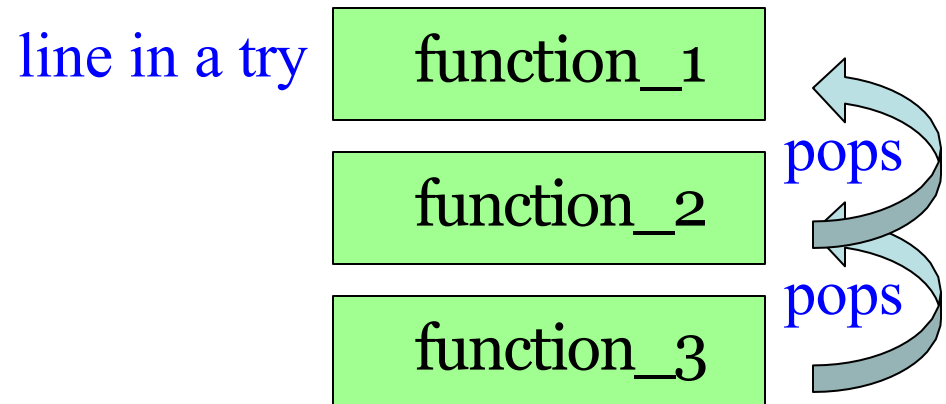
```
# recover.py
```

```
def function_1(x,y):  
    try:  
        return function_2(x,y)  
    except:  
        return float('inf')
```

```
def function_2(x,y):  
    return function_3(x,y)
```

```
def function_3(x,y):  
    return x/y # crash here
```

- Error “pops” frames off stack
 - Starts from the stack bottom
 - Continues until it sees that current line is in a try-block
 - Jumps to except, and then proceeds as if no error



Try-Except and the Call Stack

```
# recover.py
```

```
def function_1(x,y):
```

```
    try:
```

```
        return function_2(x,y)
```

```
    except:
```

```
        return float('inf')
```

```
def function_2(x,y):
```

```
    return function_3(x,y)
```

```
def function_3(x,y):
```

```
    return x/y # crash here
```

- Error “pops” frames off stack
 - Starts from the stack bottom
 - Continues until it sees that current line is in a try-block
 - Jumps to except, and then proceeds as if no error

- **Example:**

```
>>> print function_1(1,0)
```

```
inf
```

```
>>>
```

No traceback!

Tracing Control Flow

```
def first(x):  
    print('Starting first.')  
    try:  
        second(x)  
    except:  
        print('Caught at first')  
    print('Ending first')
```

```
def second(x):  
    print('Starting second.')  
    try:  
        third(x)  
    except:  
        print('Caught at second')  
    print('Ending second')
```

```
def third(x):  
    print('Starting third.')  
    assert x < 1  
    print('Ending third.')
```

What is the output of first(2)?

Tracing Control Flow

```
def first(x):  
    print('Starting first.')  
    try:  
        second(x)  
    except:  
        print('Caught at first')  
    print('Ending first')
```

```
def second(x):  
    print('Starting second.')  
    try:  
        third(x)  
    except:  
        print('Caught at second')  
    print('Ending second')
```

```
def third(x):  
    print('Starting third.')  
    assert x < 1  
    print('Ending third.')
```

What is the output of first(2)?

'Starting first.'
'Starting second.'
'Starting third.'
'Caught at second'
'Ending second'
'Ending first'

Tracing Control Flow

```
def first(x):  
    print('Starting first.')  
    try:  
        second(x)  
    except:  
        print('Caught at first')  
    print('Ending first')
```

```
def second(x):  
    print('Starting second.')  
    try:  
        third(x)  
    except:  
        print('Caught at second')  
    print('Ending second')
```

```
def third(x):  
    print('Starting third.')  
    assert x < 1  
    print('Ending third.')
```

What is the output of first(0)?

Tracing Control Flow

```
def first(x):  
    print('Starting first.')  
    try:  
        second(x)  
    except:  
        print('Caught at first')  
    print('Ending first')  
  
def second(x):  
    print('Starting second.')  
    try:  
        third(x)  
    except:  
        print('Caught at second')  
    print('Ending second')
```

```
def third(x):  
    print('Starting third.')  
    assert x < 1  
    print('Ending third.')
```

What is the output of first(0)?

'Starting first.'
'Starting second.'
'Starting third.'
'Ending third'
'Ending second'
'Ending first'

Sequences: Lists of Values

String

- `s = 'abc d'`

0 1 2 3 4

a	b	c		d
---	---	---	--	---

- Put characters in quotes
 - Use `\'` for quote character
- Access characters with `[]`
 - `s[0]` is 'a'
 - `s[5]` causes an error
 - `s[0:2]` is 'ab' (excludes c)
 - `s[2:]` is 'c d'

List

- `x = [5, 6, 5, 9, 15, 23]`

0 1 2 3 4 5

5	6	5	9	15	23
---	---	---	---	----	----

- Put values inside `[]`
 - Separate by commas
- Access **values** with `[]`
 - `x[0]` is 5
 - `x[6]` causes an error
 - `x[0:2]` is [5, 6] (excludes 2nd 5)
 - `x[3:]` is [9, 15, 23]

Sequences: Lists of Values

String

- `s = 'abc d'`

0 1 2 3 4

a	b	c		d
---	---	---	--	---

- Put characters in quotes
 - Use `\'` for quote character

- Access characters

- `s[0]` is 'a'
- `s[5]` causes an error
- `s[0:2]` is 'ab' (excludes c)
- `s[2:]` is 'c d'

List

- `x = [5, 6, 5, 9, 15, 23]`

0 1 2 3 4 5

5	6	5	9	15	23
---	---	---	---	----	----

- Put values inside `[]`

- `x[6]` causes an error
- `x[0:2]` is `[5, 6]` (excludes 2nd 5)
- `x[3:]` is `[9, 15, 23]`

Sequence is name given to both

Lists Have Methods Similar to String

```
x = [5, 6, 5, 9, 15, 23]
```

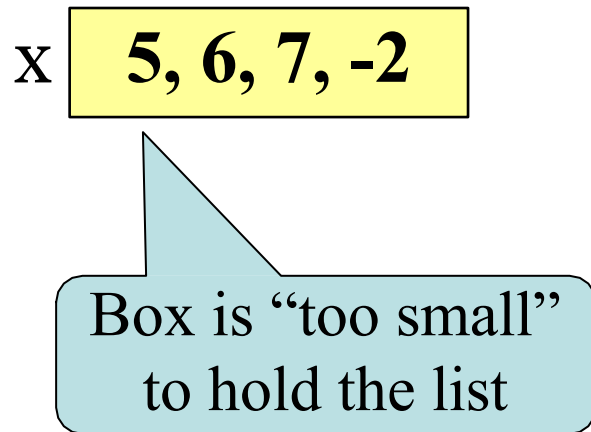
- **index(value)**
 - Return position of the value
 - **ERROR** if value is not there
 - `x.index(9)` evaluates to 3
- **count(value)**
 - Returns number of times value appears in list
 - `x.count(5)` evaluates to 2

But you get length of
a list with a regular
function, not method:

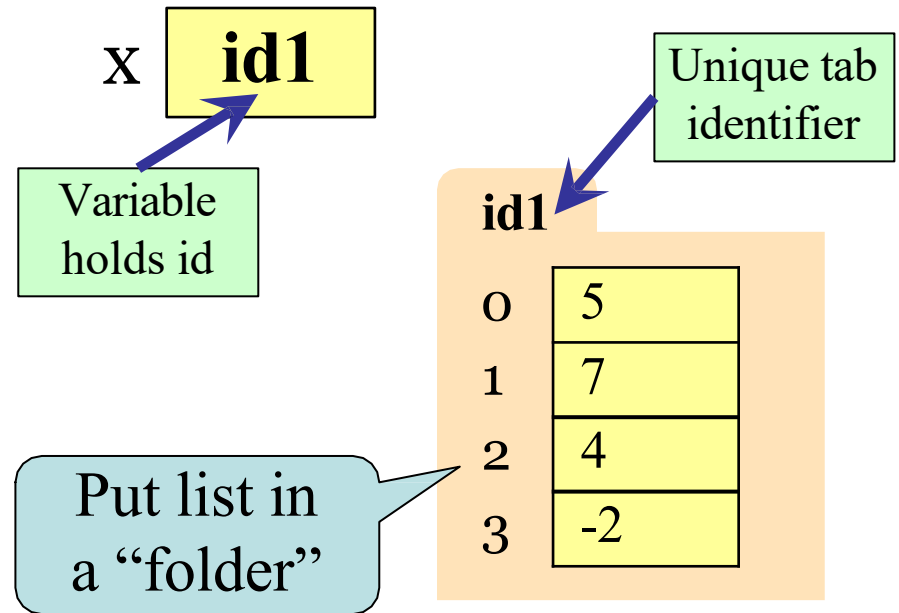
`len(x)`

Representing Lists

Wrong



Correct

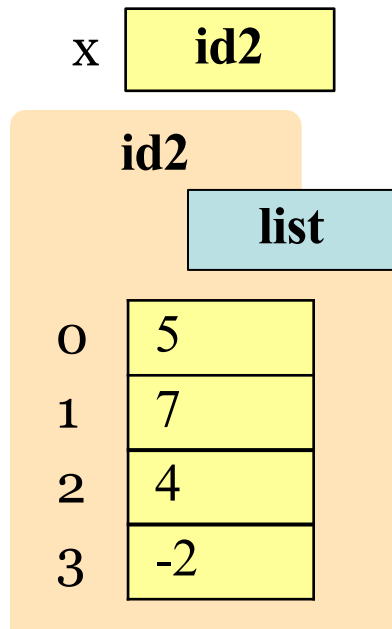


$x = [5, 7, 4, -2]$

Lists vs. Class Objects

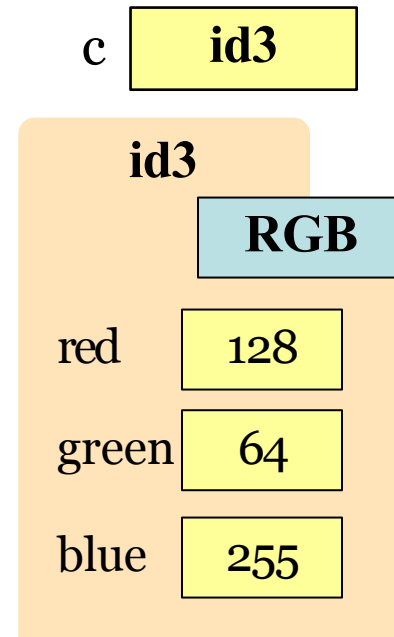
List

- Attributes are indexed
 - Example: `x[2]`



RGB

- Attributes are named
 - Example: `c.red`



When Do We Need to Draw a Folder?

- When the value **contains** other values
 - This is essentially what we mean by ‘object’
- When the value is **mutable**

Type	Container?	Mutable?
int	No	No
float	No	No
str	Yes*	No
Point3	Yes	Yes
RGB	Yes	Yes
list	Yes	Yes

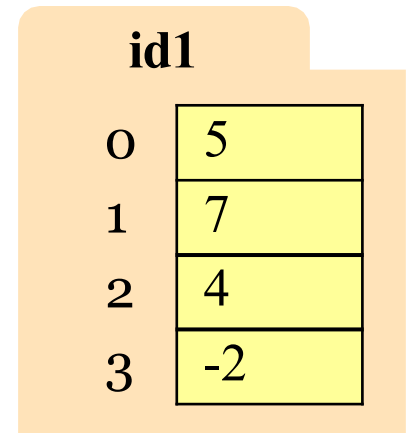
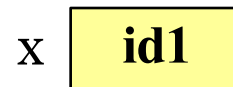
Lists are Mutable

- **List assignment:**
 $\langle \text{var} \rangle [\langle \text{index} \rangle] = \langle \text{value} \rangle$
 - Reassign at index
 - Affects folder contents
 - Variable is unchanged
- Strings cannot do this
 - $s = \text{'Hello World!'}$
 - $s[0] = \text{'J'}$ **ERROR**
 - String are **immutable**

• $x = [5, 7, 4, -2]$

0	1	2	3
5	7	4	-2

• $x[1] = 8$



Lists are Mutable

- **List assignment:**
 $\langle \text{var} \rangle [\langle \text{index} \rangle] = \langle \text{value} \rangle$
 - Reassign at index
 - Affects folder contents
 - Variable is unchanged
- Strings cannot do this
 - $s = \text{'Hello World!'}$
 - $s[0] = \text{'J'}$ **ERROR**
 - String are **immutable**

- $x = [5, 7, 4, -2]$

0	1	2	3
5	7	4	-2
8			

- $x[1] = 8$

x id1

id1	
0	5
1	7 8
2	4
3	-2

Slice Assignment

- Can *embed* a new list inside of a list
 - **Syntax:** `<var>[<start>:<end>] = <list>`
 - Replaces that range with content of list
- **Example:**

```
>>> a = [1,2,3]
```

```
>>> b = [4,5]
```

```
>>> a[:2] = b
```

```
>>> a
```

```
[4, 5, 3]
```

Replaces [1,2]
with [4,5]

List Methods Can **Alter** the List

```
x = [5, 6, 5, 9]
```

- **append(value)**
 - A **procedure method**, not a fruitful method
 - Adds a new value to the end of list
 - `x.append(-1)` *changes* the list to [5, 6, 5, 9, -1]
- **insert(index, value)**
 - Put the value into list at index; shift rest of list right
 - `x.insert(2,-1)` changes the list to [5, 6, -1, 5, 9,]
- **sort()**

List Methods Can **Alter** the List

```
x = [5, 6, 5, 9]
```

- **append(value)**
 - A **procedure method**, not a fruitful method
 - Adds a new value to the end of list
 - **x.append(-1)** *changes* the list to [5, 6, 5, 9, -1]
- **insert(index, value)**
 - Put the value into list at index; shift rest of list right
 - **x.insert(2,-1)** changes the list to [5, 6, -1, 5, 9,]
- **sort()**

What do you think this does?

Where To Learn About List Methods?

5.1. More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.append(x)`

Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.extend(iterable)`

Extend the list by appending all the items from the

In the documentation!

`list.insert(i, x)`

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.remove(x)`

Remove the first item from the list whose value is equal to `x`. It raises a `ValueError` if there is no such item.

`list.pop([i])`

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

Lists and Functions: Swap

1. `def swap(b, h, k):`
2. `""" Swaps b[h] and b[k] in b`
3. `Precond: b is a mutable list,`
4. `h, k are valid positions"""`
5. `temp= b[h]`
6. `b[h]= b[k]`
7. `b[k]= temp`

`swap(x, 3, 4)`

swap		5
b	id4	h 3
		k 4

Swaps `b[h]` and `b[k]`,
because parameter `b`
contains name of list.

id4

0	5
1	4
2	7
3	6
4	5

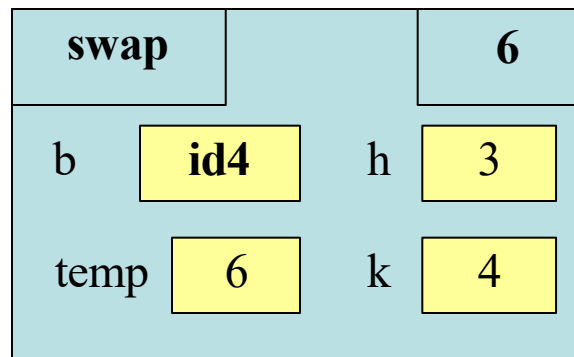
x

id4

Lists and Functions: Swap

1. `def swap(b, h, k):`
2. `""" Swaps b[h] and b[k] in b`
3. `Precond: b is a mutable list,`
4. `h, k are valid positions"""`
5. `temp= b[h]`
6. `b[h]= b[k]`
7. `b[k]= temp`

`swap(x, 3, 4)`



Swaps `b[h]` and `b[k]`,
because parameter `b`
contains name of list.

id4

0	5
1	4
2	7
3	6
4	5

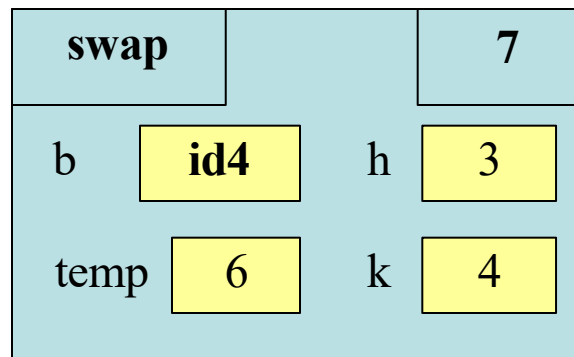
x

id4

Lists and Functions: Swap

1. `def swap(b, h, k):`
2. `""" Swaps b[h] and b[k] in b`
3. `Precond: b is a mutable list,`
4. `h, k are valid positions"""`
5. `temp= b[h]`
6. `b[h]= b[k]`
7. `b[k]= temp`

`swap(x, 3, 4)`



Swaps `b[h]` and `b[k]`,
because parameter `b`
contains name of list.

id4

0	5
1	4
2	7
3	6 5
4	5

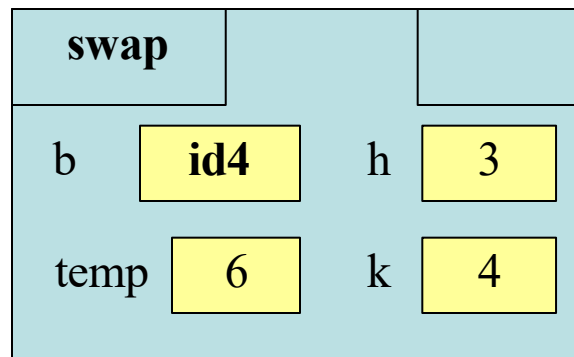
x

id4

Lists and Functions: Swap

1. `def swap(b, h, k):`
2. `""" Swaps b[h] and b[k] in b`
3. `Precond: b is a mutable list,`
4. `h, k are valid positions"""`
5. `temp= b[h]`
6. `b[h]= b[k]`
7. `b[k]= temp`

`swap(x, 3, 4)`



Swaps `b[h]` and `b[k]`, because parameter `b` contains name of list.

id4

0	5
1	4
2	7
3	6 5
4	5 6

x

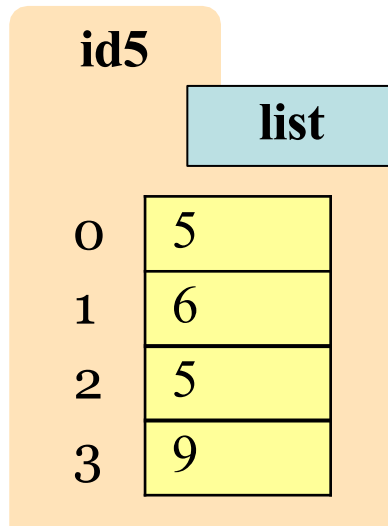
id4

List Slices Make Copies

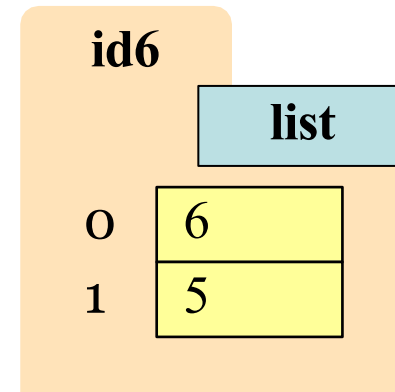
$x = [5, 6, 5, 9]$

$y = x[1:3]$

x **id5**



y **id6**



copy = new folder

Exercise Time

- Execute the following:

```
>>> x = [5, 6, 5, 9, 10]
```

```
>>> x[3] = -1
```

```
>>> x.insert(1,2)
```

- What is x[4]?

A: 10

B: 9

C: -1

D: **ERROR**

E: I don't know

Exercise Time

- Execute the following:

```
>>> x = [5, 6, 5, 9, 10]
```

```
>>> x[3] = -1
```

```
>>> x.insert(1,2)
```

- What is x[4]?

-1

- Execute the following:

```
>>> x = [5, 6, 5, 9, 10]
```

```
>>> y = x[1:]
```

```
>>> y[0] = 7
```

- What is x[1]?

A: 7

B: 5

C: 6

D: **ERROR**

E: I don't know

Exercise Time

- Execute the following:

```
>>> x = [5, 6, 5, 9, 10]
```

```
>>> x[3] = -1
```

```
>>> x.insert(1,2)
```

- What is x[4]?

-1

- Execute the following:

```
>>> x = [5, 6, 5, 9, 10]
```

```
>>> y = x[1:]
```

```
>>> y[0] = 7
```

- What is x[1]?

6

Lists and Expressions

- List brackets `[]` can contain expressions
- This is a list **expression**
 - Python must evaluate it
 - Evaluates each expression
 - Puts the value in the list
- Example:

```
>>> a = [1+2,3+4,5+6]
>>> a
[3, 7, 11]
```
- Execute the following:

```
>>> a = 5
>>> b = 7
>>> x = [a, b, a+b]
```
- What is `x[2]`?

A: 'a+b'

B: 12

C: 57

D: **ERROR**

E: I don't know

Lists and Expressions

- List brackets [] can contain expressions
- This is a list **expression**
 - Python must evaluate it
 - Evaluates each expression
 - Puts the value in the list
- Example:

```
>>> a = [1+2,3+4,5+6]
>>> a
[3, 7, 11]
```
- Execute the following:

```
>>> a = 5
>>> b = 7
>>> x = [a, b, a+b]
```
- What is x[2]?

12